AD-A104 924 BATTELLE COLUMBUS LABS OH F/G 9/2
TESTING OF DATA PROCESSING SOFTWARE.(U)
JAN 80 R HAMLET DAAG29-76-D-0100
UNCLASSIFIED NL

1 OF 1
AD A
104924

END
DATE
FILMED
10-81
DTIC

LEVEL

(15)

Testing of Data Processing Software.

~~Prepared under~~
Contract DAAG29-76-D-0100, new
Delivery Order 1322, for
Battelle Columbus Laboratories

Final Report
Jan~~uary~~ 1980

by

Richard Hamlet

The views, opinions, and/or findings contained in this report are those of the author, and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other documentation.

81 [illegible] 401000

Table of Contents

1. Scope and Purpose *1*

2. Testing and Testing Tools *2*
   - 2.1 The Art of Testing *2*
   - 2.2 Controlling and Documenting Tests *3*
   - 2.3 Utility Tools *4*
   - 2.4 Exercising Tools *6*
   - 2.5 Impractical Testing Ideas *8*

3. Computer Systems Command Testing *10*
   - 3.1 Existing Practices *10*
   - 3.2 Previous Studies *11*
   - 3.3 Assumptions and Viewpoints *12*
   - 3.4 Proposal Summary *13*
   - 3.5 Use of Tools *14*
   - 3.6 Practices and Training *17*
   - 3.7 Quality Assurance *19*

Annotated Bibliography *24*

Summary

This report surveys the field of software testing, and suggests applications to the testing practices of the United States Army Computer Systems Command. Its emphasis is on the practical aspects of testing, as they are seen by working programmers. Other studies which have addressed management issues are briefly examined.

The study is addressed to the actual Computer Systems Command environment. Therefore, a good deal of the research literature on testing is irrelevant. A few comments that characterize impractical research are included to give direction to those examining the literature. Recommendations whose implementation would require substantial changes in resources have been avoided. It is assumed that the Computer Systems Command can move gradually to a better employment of its testing effort, but that no substantial increase in the resources devoted to testing can occur.

Briefly, the report concludes:

1. A number of practical automatic testing tools exist, and the Computer Systems Command should be using them at all levels. In some cases this will require development of the tools for the Command environment.

2. Testing practices should be more formally defined within the Command. More should be written about them, and they should be the subject of formal training.

3. The problems of maintenance testing are more easily defined and attacked than those of new-development testing. The Command should make this distinction more than it now does, and concentrate on the easier, maintenance problem.

4. The role of the Quality Assurance group within the command should be altered so that instead of actually carrying out tests, it oversees and certifies the tests done by others.

1 Scope and Purpose

This study has two purposes, to survey the art of software testing today, and to apply it to the data processing effort of the Computer Systems Command. The study was confined to the practical. A great deal of computer science research might be used today by an organization willing to take the risk of developing and utilizing tools and techniques that have promise rather than a proven record. However, if the early history of computer applications is any guide, the first organizations to make this gamble will provide invaluable experience for their successors who buy them out at bankruptcy. Although this report contains some characterizations of what is and is not practical, it makes no attempt to discuss or cite those ideas that cannot be safely and immediately applied. Similarly, the study assumes that the Computer Systems Command is a stable organization whose resources and methods can change only gradually. Where recommendations are made, care has been taken to make them realizable without substantial alterations in responsibilities, personnel, or other resources.

However, the emphasis on practicality and existing resources does not preclude rather sweeping changes in principle. In many cases the Command can make a small change and a small commitment that will have immediate beneficial results, and will later point in a direction of larger changes. Such shifts in emphasis and viewpoint should not be judged for what they might accomplish or might cost if carried to some ill-defined extreme, but rather should be judged on their immediate benefit. Several of the recommendations have exactly this character. For example, to view program maintenance and its testing as an entirely different problem from program development and its testing, is a fundamental change. But it can be begun simply by using tools and techniques that work for maintenance but might not work so well for development, with immediate benefit. If in the end the separation of maintenance and development leads to some far-reaching changes in the Command, those will grow naturally and only if they are beneficial. Meanwhile the immediate gains have been made.

An author's background and interests influence what he believes and writes. His background also influences how he is read. A report on the safety of historical London streets might be taken rather differently if its author were Jack the Ripper rather than Robert Peel. I therefore want to describe my own bias at the outset. My academic interest is in the theory of automatic testing as an alternative to formal verification proofs. However, I come from a background of systems programming, in particular several years as supervisor of a medium-scale system responsible for administrative data processing of a large university. I think that there is promise in testing methods for the future; at the same time I believe that the best methods today are very flimsy weapons against the problems of real software.

2 *Testing and Testing Tools*

It is impossible to conduct any software test without computer assistance, but the extent to which the computer is used merely to execute the program under test, or to itself aid and control the test, varies greatly. There is no better way to insure that a valuable testing practice is actually used, than to make it automatic and machine implemented. The analogy to programming-language compilers is a useful one: because the source language is specified, the compiler must be used; syntax errors are mechanically detected, and must be eliminated. A testing tool that functions in the same way has the same advantages: it uses computer time to save people time, and to introduce standardization that cannot be avoided. Sections 2.2 - 2.4 describe practical testing tools under three broad headings. (In Section 2.5 there is a brief discussion of ideas that are less useful.) The questions of test training, administration, and certification can be viewed as technical, rather than managerial questions from the viewpoint of tools. If the right tools exist, the management problem is solved by requiring their use; the other aspects also follow naturally. It is not claimed that the tools described here are a final answer. But they are useful at small cost, and their use will effect a definite improvement in testing effectiveness.

2.1 The Art of Testing

Even small computer programs have a vast number of "states"—the combination of internal storage and program control points that the program employs to remember information about the input it has already seen, and the progress of its computation. It is this large number of states that makes program testing a hopeless business in practice and in principle: an exhaustive test would have to put the program in each of its possible states, and then supply it with all combinations of input that might occur there. Any test short of this exhaustive one is untrustworthy because it could happen that the program contains an error that shows itself in just the circumstances that have not been tried. Exhaustive testing is not practical, so what is the point of program testing at all? If programs were created randomly there would be none—an attempt to successively correct a program generated at random to one that solves some useful problem would not succeed. But programs are not written by random trial and error, and practical testing relies on the premise that the program under test is "almost right." If the errors are relatively few, if they are errors of mistaken commission rather than omission, if they are so placed that many inputs expose them, rather than requiring a peculiar, unique state in which to appear, then most of them can be found by testing. Thus the first part of the testing art is not testing at all, but the art of programming, and the good fortune to make mistakes that are glaring rather than subtle. The implication is two fold: testing should begin as soon as possible in the programming process, and should be done by those with the most knowledge of a program, while that knowledge is fresh.

It is observed that some people are better at program testing than others, and that no simple set of characteristics separates the good from the bad. Thus we say that testing is an art rather than a science, and those who have more success have mastered this art. Like any art, good testing can be learned by observing someone who is successful, but only imperfectly. There are some obvious principles involved, and these can be formally taught. For example, it is *important* to keep good records, to notice what a test does *not* prove, to concentrate on cases that are likely to be forgotten in programming, to view success as *finding* errors, etc. Furthermore, much of good testing is routine but exacting work; here the computer can be used to handle mechanical detail.

There is every reason to expect that a combination of adequate training in testing practice, appropriate timing of tests, and use of automatic testing tools will make a significant improvement in test effectiveness. On the other hand, there is no reason to expect that the best testing plan will approach perfection. Software subjected to the best tests available, flawlessly applied, will still contain bugs, and will still go into the field in embarrassing condition. But with proper testing the incidence of errors can be reduced, and the cost of finding and fixing them can be lowered. Every time a test finds a problem and causes it to be fixed *before* release of the software, the saving in field trouble and avoiding the maintenance mechanism is immense. We should not hope to eliminate post-release problems, but to reduce their severity, and allow the maintenance organization to work more on real changes and less on old bugs.

2.2 Controlling and Documenting Tests

It is attractive to believe that there is no more to program testing than carefully controlled trial and error. Software is tested (never mind how) and released. It fails in the field on particular data. That data is added to the test so that when the next release occurs, it cannot fail again in the same way. This process continues over a long time so that eventually the tests are so extensive, and so like the actual environment the released software will face, that most bugs are caught before release. (It is evident that this view only makes sense when the software changes slowly and its field application remains almost static. However, that is exactly the situation with a good deal of standard data processing software.)

For a large system manipulating substantial files, it is an important problem to manage its test data. The part of the problem that involves modifying and creating test data is considered in the following section on utility tools. Here we consider the management and control of tests. Test data should be viewed very like program sources: it has versions and releases, correlated with the sources and with field reports and change requests. Just as it is important to keep track of source program changes and releases, so it is important to know which test is designed for which purpose, and to use this information from past tests in the future. For example, a base test might be initially constructed from actual field data. Later, a bug might be exposed by some special, unusual data from the field. There is every reason to add this case to the base, and to use it in all future tests. On the other hand, a test specially designed to see if internal tables handle size overflow correctly should probably be filed away and seldom run. It should not be added to the base test. The example shows that a given system may have many different tests associated with it, and each has a purpose. Given unlimited time and resources, it would be wise to run every test every time the software is changed. In reality there should be selection of the best tests, and this requires systematic filing and description.

The tools that aid in test documentation are similar to those used to handle program releases and changes. Cycles and versions must be maintained, with a history sufficient to show when a later test replaces a former, and the relationship between them. It would be ideal if such a test maintenance scheme were integrated with program, requirements, and history documentation. It might then be appropriate to try to establish links among requirements, programs, and tests so that inquiry could be made about the connections (or lack of them). For example, if a system fails in the field following a maintenance release, it would be valuable to find out why the release test did not catch the problem, and this would begin by inquiring what tests were added, related to which program changes. Such an integrated system lies in the future; however, there is no reason not to mechanize and store test descriptions now.

2.3 Utility Tools

If an editor is the primary tool of program development in an interactive environment, a file-comparison program is the corresponding testing tool. Such a program displays the differences between two files in a revealing format. Its most obvious use is comparing a pair of output files produced by different versions of the same system, which should be the same. However, comparisons have many other uses in testing. They can be used on input files to verify that a few special records have been inserted in the proper place. They can be used on source programs to graphically display differences that must be tested. (The latter comes up again in Sections 2.4 and 3.5 on program exercising.)

The more special features a comparison program has the better. It may be possible to select portions of files, or to force dissimilar files into "register" at a given point, from which the comparison proceeds. It is useful to be able to place the difference output in a file for later processing. Often special features can be simulated with an editor in conjunction with a rudimentary difference program. Ability to format the differences in many ways is indispensible. Comparisons that assume a particular file format are usually less useful, although powerful when that format exists.

The essence of comparison processing is to minimize exceptions. The best comparison is one in which no differences must be examined. For example, to see if an updating program is properly inserting a record, the following sequence might be used: (a) prepare two inputs that differ by the record, and compare them, saving the difference (the record); (b) compare the output files from these inputs, saving the difference; (c) compare the differences saved in (a) and (b). Success is indicated by no differences in (c). The example is oversimplified, but such a scheme is obviously superior to human examination of the output file, particularly when the data volume is high.

Brute-force comparisons of large, dissimilar files can be expensive, but some recent algorithms may help [Heckel, 1978]. Vendors seldom supply comparison programs, perhaps because they don't like to acknowledge the problems for which they are needed.

File dump and edit programs that operate at the lowest (bit, character) level are seldom necessary if comparisons are used creatively. When dumps are needed, it is often important that they *not* automatically process records as formatted entities. It may be erroneous assumptions about record formats that cause tests to miss program errors (because the test processing makes the same false assumptions made by the programs under test). A raw dump of (say) a magnetic tape, with all record sizes, file marks, etc., displayed, often cuts through such misconceptions. The ability to edit files at the lowest level is also useful, because it can be used to easily force a program to take an unusual action without creating a vast complex of data. For example, adding a field to a record in the middle of a long sequence may be the best way to resolve a question about what a program is doing; a low-level editor can do this much more easily than a program designed to build such files in their entirety.

For a system with a long lifetime, it is worthwhile building special-purpose tools to create, modify, and evaluate test data peculiar to the system. Such tools can be interactive and can easily create low-volume tests. The program tester is often placed under pressure to complete his work, and has no time to develop tools that would make it much easier and more accurate. He gets through the test on time by putting in long hours, only to find himself facing the same situation on the next release. It is false economy not to invest in the necessary tools. The system itself may aid in developing special test tools At the least, the COBOL data division can be copied to describe the necessary files, and routines for formatting output can be taken over without much change. The Test Data Generator facility of MetaCOBOL [ADR, 1974] is an automated aid to such program-specific testing, and a good one. Its drawback is that it requires skill to use successfully.

A cleverer approach actually builds testing code into the source program, under conditional compilation control. With the testing code turned on the program can perform internal checks, but it can also be used to manipulate test data. For example, following an input phase there can be code inserted to print a summary of the record types in the input file. In production use this code will not be compiled, but it remains with the source so that it can be turned on and used to test each new release. A way to control and implement conditional compilation is with the COPY and INCLUDE options of COBOL—the files copied can be empty for the production compile.

2.4 Exercising Tools

In the 1970s there was considerable interest shown in automatic, "structural" testing of programs. This idea is no more than the construction of automatic processors to do what a person might do to be sure tests actually "exercise" code. These methods have no theoretical justification—it is easy to come up with examples of completely exercised programs that nevertheless fail badly—but it seems to work in practice. The test is tied to the structure of the program under test, and the "coverage" is measured against that program. Some systems generate the data to force coverage, others merely record the coverage of a set of data supplied by the person doing the testing.

The simplest coverage measure is that of executed statements. If a program is tested, yet certain statements have never been executed, then an arbitrary bug may be lurking in those statements. To pass through each statement does not guarantee that all is well, for the usage may not be the crucial one on which a statement fails, but it is evidently better than not passing through some at all. A "statement exercising" test tool would take as input a source program and generate test data *to force the execution* of each statement; or, it would take a program and a collection of tests as input, and report whether or not those tests did execute all statements.

Exercisers that generate tests are far more complicated than those that monitor supplied test data. Furthermore, the generation of tests is more difficult when the source program has complex patterns of control (is not "structured") or uses complex tests to determine flow of control. Most test-generation research has been done on FORTRAN or similar "scientific" languages. The remainder of this section is largely devoted to exercisers that report test coverage rather than generate test data.

To monitor a test requires only the simplest of execution-time data gathering, and a program instrumented to produce a report on (say) statement execution coverage runs almost as quickly as an uninstrumented program. A final characteristic of structural coverage is the form in which it can be reported: failures are specific and local, and call programmer attention to the parts of the program where errors may lie. The obvious response is to generate more data to cover those parts, or to modify the program. While this response is not mechanical, the situation is rather like that of correcting syntax errors in response to compiler messages. It may still be that the program will not succeed, but the chance has been improved at small cost.

Statement exercising is the simplest form of a more general case called path exercising. It is more stringent to insist that every path in a program be traversed by some test data than to ask only that each statement be executed. The number of paths may be unbounded, if each pass around a loop is considered a distinct path. Exhaustive path testing of large programs becomes prohibitively expensive, and is not necessarily more reliable than statement exercising. A number of in-between ideas have been tried, including trying both alternatives of each conditional expression (branch testing), and trying some finite number of repetitions of each loop (level-I testing). Perhaps the most theoretically attractive is the testing of "basis paths," those adequate to build up all other paths. The technology exists today to use any of these schemes automatically with any source language, although few analyzers have actually been built for COBOL. There is no question but that the schemes work at the unit testing level, and that some of them (e.g., branch testing) are worth their cost.

The deficiency in path testing is that it gives no measure of whether the use of a path was "significant." For example, passing through the statement
?

COMPUTE A = B*(C+D)

with a zero value for B isn't much of a test—an arbitrary bug could be hiding in the remainder of the expression. Systems have recently appeared that attempt to remedy this fault, and provide a kind of "expression testing" that is orthogonal to path coverage. Again, although there is no theoretical justification, these systems seem to work to find errors, and are specific and local in the potential difficulties they report. The simplest expression coverage scheme detects only that test data has failed to make an expression change its value. (In the example above, so long as B is zero, the expression B*(C+D) takes no value but zero; one might as well have written MOVE ZERO TO A.) All expression exercising is more expensive and less easy to understand than path testing, but perhaps it is correspondingly more revealing. In any case, the simplest schemes are worth their cost.

Program exercisers have a dual character. On the one hand they are a tool for programmers to use in finding errors. When the exerciser flags part of the program as untested, the programmer is led to try more tests or fix a bug. But on the other hand, exercisers are also measurement tools: they can be used to quantify the quality of tests by their coverage. It takes skill and knowledge of the program for the first use; the second use requires only examining records to determine the results. It is therefore appropriate for these tools to figure heavily in a formal test plan, to find bugs, and to help audit the test [Sorkowitz, 1978]. Exercisers have the further virtue that they can be used at all skill levels. A clever programmer will employ them to find many bugs (and may even adjust his coding practices to rely on the tool and concentrate on things it cannot test). A poor programmer will get little advantage from it, but will not thereby waste time—if the tool doesn't help him, he can ignore its messages and use whatever techniques he would have used anyway.

The expense of path and expression testing grows roughly as the square of the size of the program (and of course directly with the number of tests, but those tests would be conducted anyway). Automatic methods can therefore become prohibitively expensive for large programs. (With the present technology, "large" is on the order of thousands of

COBOL statements, not very large in practice.) An order of magnitude improvement in efficiency can be obtained by building the testing tool into a compiler so that the tested code is compiled rather than interpreted [Hamlet, 1977]. Another scheme for lowering the cost is to apply the coverage to only a randomly-chosen fraction of the program. (Theoretically, it seems better to do some kind of simple coverage on the whole than something complex on only part, however.) Fortunately there may be efficient ways to exercise large, long-lived systems. In the maintenance situation, the number of lines of code *changed* is typically far smaller than the total size of the program, and in many cases it is those lines that must be exercised, not those which are unchanged. Here even automatic generation of test data is practical [Foster, 1978]. For example, if a change is limited to a few formulas (as statutory changes may be), it is possible to perform almost an exhaustive test of those formulas at small cost.

The best use of available exercising tools is still a subject of research, but the tools are ready to carry simple, proven schemes into practice. Programmers are as likely as researchers to find new ways to use them.

2.5 Impractical Testing Ideas

It is important to distinguish among several kinds of "impracticality" in judging ideas for programming. There are at least three kinds:

(I) An idea is impractical because it is too poorly defined to use. In this case it is difficult to know if the idea is being used or not, and honest people can forever disagree about its merits. Many popular "buzz words" describe such ideas. For example, "modularity" certainly means breaking programs into subunits in some way. But without a much better definition of what characterizes the "module," the idea is not a practical one. Often a poorly defined idea gropes toward something of importance, and in that case it can be expected that it will gradually become better defined, and may in the end be very practical. ("Modularity" is probably such an idea, if it can be linked to maintenance so that only a few modules need to be altered when a change is made. The ill-defined "modules" of today do not have this property.)

(II) An idea is impractical because it is too hard for most people to understand. It may not be possible to teach even a good idea to most programmers for their productive use. Ideas of this kind may be defined by results, not by a systematic procedure to gain those results. Another common characteristic is that extensive theoretical background and practice is needed to use the idea. Program verification is the best example of such an impractical idea. It requires some knowledge of formal logic to even understand what verification proofs are all about, and even then the theory tells more how to recognize a proof than how to construct it. Some hard ideas have worked their way down from impracticality to everyday use, but because someone discovered a way to apply the theory without understanding it. (In electrical engineering, the use of transforms for circuit analysis is an example. Engineers can use the transforms to solve very difficult problems without having to acquire the insight of a Fourier or Heaviside.) "Symbolic execution" may be an example of an idea now approaching practicality. Although the ideas behind symbolic execution systems are not shallow, these systems may soon be easy to use without a deep understanding [Darringer and King, 1978].

(III) A programming idea may be impractical because it makes inefficient use of computer resources. How important this deficiency becomes depends on the magnitude of the inefficiency. If a technique takes ten times too much machine time to carry out, there is a good chance that a faster machine or algorithm can improve the performance so that it becomes practical. On the other hand, if the time factor is 10,000, only a new, hard-to-discover algorithm may be practical. Many problems are "combinatoric" in nature, and algorithms significantly faster than exponentially-growing trial and error are not known. High-level languages and their compilers are an example of the first situation. Initially they were too slow, and produced code that wasn't fast enough. (Some say that the situation persists today for real-time applications.) But faster machines and some clever ideas soon overcame the small factors involved, and today no one would call high-level languages impractical. As an example of the second situation in which there is apparently no hope for practicality, consider exhaustive testing of programs. A one-page program taking a single integer value as input on a 32-bit machine might require a millisecond for one execution; there are about a billion possible inputs, so the exhaustive test time would be about 30 years (not counting, of course, the time to examine the output to see if it is correct). And these numbers rise exponentially with program and data complexity.

Most programming ideas suffer from a combination of the three kinds of impracticality, constantly changing under the impetus of new technology, rising skill levels, and new research. Indeed, the whole idea of automatic data processing was certainly impractical in all three ways only 40 years ago. Thus the question of impracticality must be constantly reexamined. On the other hand, some promising ideas never become practical. The only sure lesson that can be learned is that it is suicidal to confuse the promise of an idea with its immediate practicality. What has happened over and over in computer applications is that an impractical idea was prematurely adopted, and in the course of its failure, it made large strides toward practicality. Unfortunately, its developer was seldom in a position to take advantage of the situation, since the initial failure removed him from the scene.

3 Computer Systems Command Testing

The recommendations for improvement of Computer Systems Command testing practices appear in Section 3.4, and are treated in detail in following sections. The reader familiar with current practice and previous studies can skip to Section 3.3.

3.1 Existing Practices

This section is included only to make this document self-contained. It describes only maintenance procedures, and only as referred to in the sections to follow. For more detail, the summary in [Fox and White, 1979] is easier to read than the regulations.

A software system existing within the Computer Systems Command is the continuing responsibility of its developer. When the system is running in the field, change requests are accumulated against it, requesting bug fixes or functional alterations. The change cycle is time-driven by predetermined release points, so the developer assigns programmers and analysts to work on changes up to a cutoff date (perhaps a few weeks prior to a release point). As the work proceeds, informal tests are conducted by the programmers, the so-called level-I and level-II Developmental Center Tests (DCT). (Level I is on single changed programs; level II on a group of interacting changed programs.) Work accomplished by the cutoff date enters the final developer's test, the level-III DCT. Here there is a formal Quality Assurance involvement, and a formal test plan. The developer uses hardware and operating systems which are not the same as those in the field, although in DCT III an attempt is made to come closer to field conditions. The result of a successful DCT III is a complete system ready to release to the field. The developer not only retains sources and documentation for the system, but also keeps test data, which will likely be used again for a forthcoming release.

The ready-to-release software now enters an independent, Environmental Test (ENT) conducted by Quality Assurance under conditions as close to those in the field as possible. However, the ENT is not a functional test—results must be obtained, but they are not examined in detail. Serious difficulties encountered during ENT may send the software back to the developer; if it can be made to pass, it enters the Field Validation Test (FVT).

A selected field installation conducts the FVT, using live data, but in parallel with an existing version of the system. Should difficulties be encountered, the developer may have to being again at an appropriate DCT level. It is in the FVT that the final functional analysis is performed, where it is certified that the requested changes have in fact been made in the software.

3.2 Previous Studies

Three recent studies of Command testing practices have been very useful in preparing this report. The two studies conducted by the Army itself contain more information about present practices than an outsider could easily obtain under time pressure.

SAI Comsystems Study [SAI, 1977]

The primary recommendation unique to this study is that the Command attempt to substitute an activity-driven mode of operation for a time-driven one. There is certainly merit in the suggestion. As the maintenance situation now stands, it is timed releases that control the time available for testing, since programming continues from just after the last release until a few weeks before the next is due. Then testing must fit into a predetermined time slot, and this causes it to be rushed and without flexibility when things go wrong. (Furthermore, there is the problem of having to do all the testing for several systems at once, overloading the machines.) However, such a change is by no means a small one, nor is it primarily technical, and so falls outside the scope of this study.

Although the point is not followed up with a technical recommendation, the SAI Study does focus on the statistical nature of testing, to make the point that although perfection is impossible, any degree of controlled imperfection can be attained. (Had there been a technical recommendation it might have been that the Command begin to use statistical methods of error frequency analysis.) The technical work on which the statistical theory rests has proven useful in hardware testing and quality control in manufacturing of hard goods. However, its validity, its practicality, and its further development are all quite controversial for software, and it would certainly be premature to apply it to data processing software. My own belief is that the theory shows little promise, and that it is likely to remain in limbo. (A General Research Corporation study [GRC, 1979] seems to agree with my assessment of statistical reliability theory, but it in turn is concerned with error-data analysis, in my opinion equally unpromising.)

Draft AIRMICS Study [Daniel and Mitchell, 1976]

The recommendations of this study (in draft only—it was never released) are similar to those presented here, but from a time perspective three years earlier, and an insider's viewpoint. The primary differences are that some testing tools are now more practical than they were, and procedures within the Command are better understood. The AIRMICS study is valuable because it includes interviews with technical people, thus broadening the data base for these conclusions. It is also interesting to note that three years has had little effect on the technical practices within the Command: testing is still little understood, and the available tools are still not used. It is clear that something must be done to begin the process, that testing procedures will not improve by themselves.

Quality Assurance Study [Fox and White, 1979]

The most recent study was conducted by a committee within the Quality Assurance organization. This study was directed more at middle management practices than at technical considerations, in particular at the role of Quality Assurance. The study is valuable because it provides an excellent description of current practices, and relates potential changes to regulations as well as to resources and goals. Its conclusion that changes can probably be made without extensive rewriting of basic regulations is comforting. The study proposes a shift in the role of Quality Assurance similar to that recommended here.

In summary, previous studies remain valuable in describing problems and practices in the Command. Perhaps some conclusions are more sweeping than can be easily implemented. The technical picture is more hopeful today than in even the recent past, and the technical emphasis of the present study complements earlier work.

3.3 Assumptions and Viewpoints

The proposals to follow in Sections 3.4 - 3.7 are based on a number of assumptions. Although these seem unexceptionable, they should be listed at the outset.

Assumptions

The regulations that control software development and maintenance within the Computer Systems Command will in no way influence alterations in testing practices suggested here. (Indeed, the regulations do not seem to say much about the technical details of testing.)

Only COBOL, data processing systems are considered here. Although the Command uses some executive software written in assembler, and is technically responsible for its maintenance, this software is stable and reliable.

Development sites within the Command do not differ significantly in their testing needs. There are significant differences in the activities at sites, but there is no reason why the best practices should not be culled from each and uniformly applied.

Although machine and personnel resources are always tight in any software project, there is sufficient machine time available (particularly on the development systems in the early stages of testing) to begin the use of automatic tools. More machine power is needed in any case, and if it is acquired, the use of test tools may expand.

Viewpoints

Two changes in viewpoint underlie most of the recommendations to follow. The practical suggestions do not require a shift in viewpoint, but may lead to one gradually.

First, program development and program maintenance are so different where testing is concerned that the Command can make a distinction. A technique that will work for maintenance can be adopted even if [illegible]s of no use in new development. This is not to imply that emphasis is misplaced on [illegible] systems, but only that the technical suggestions here apply to the maintenance case. Testing is radically different for new work and for program changes because in the former case there is no base of tests and experience to draw on, and the requirements of the test are far more difficult. For example, the proponent agency has a large role in interpreting ill-understood specifications, which is largely absent in the maintenance case. Furthermore, much maintenance deals with only relatively small blocks of code (compared to the volume of the whole system), and the tools for testing work much better on small parts of programs than on larger ones.

Second, it is suggested that independent, third-party testing is only a good idea if it is an oversight activity. In gaining the independence of a third-party test, so much expertise is lost from the proponent and development groups, that the results are bias-free, but too shallow to be worthwhile. Only by keeping those who know most about the software in the testing loop can their deep knowledge be used, not only to perform better tests, but to speed up the cycle of error-detection and correction. While there is always danger of (often unconscious) bias when the test group has special knowledge, this danger is less than the one of trivializing the tests. Wherever objective measures of test quality are available, the best of both worlds can be obtained: the expert group can efficiently conduct the test, and the independent group can verify (in the sense of an auditor) that the objective measure is satisfied. Here the analogy with compilation is again useful. It would assure that syntax errors do not occur in released code to leave all compilations to an indepencent quality assurance group. But the compile-and-fix cycle would lengthen ridiculously, and it is just as good for the independent group to check listings (or better still, machine records that show the listing to be error-free).

3.4 Proposal Summary

The following recommendations are described in detail in the sections to follow:

1) The Computer Systems Command should be using more automatic testing tools. In the creation and maintenance of test data there should be conventional editing and library facilities, which should be used not only to ease the burden of updating test data, but also to record and document testing decisions, and to relate them to field reports and program changes. Programmers should use automatic exercising tools, particularly in the early stages of testing, and with particular emphasis on the parts of code that have been altered in maintenance. All tests should routinely employ automatic comparisons to verify and document that files are preserved or that appropriate changes have occured. Some of the necessary tools already exist at scattered sites; some can be easily developed as a part of routine programming activity within the Command; some will require more formal research activity, perhaps in AIRMICS. (See Section 3.5)

2) The Command should make a concerted effort to codify the best testing practices

that now exist, and to continually add to these as technology and expertise develop. Some of this effort can be done in the form of regulations, but since technology changes so rapidly, perhaps a more flexible device is needed. Formal training in testing practices should be routinely and systematically provided, but the training program should initially be designed outside the Command. Whenever a practice has been shown to be beneficial, it should be required at the appropriate level by writing its use into test plans, requirements, or regulations. Sites should not be permitted to develop too many local peculiarities: if a local practice is good it should be extended, or if not, dropped. The responsibility for continuing oversight of test practices should fall to Quality Assurance. (See Section 3.6)

3) The Quality Assurance testing effort should be converted to a passive, oversight role. Insofar as testing activities become standardized and mechanized, the certification of appropriate testing is routine. Quality Assurance should seek to reduce its testing oversight role to this desirable state, and where that is impossible, to develop informal guidelines to improve its art. In brief, Quality Assurance should retain responsibility for software passing its tests, but others should conduct the tests. (See Section 3.7)

3.5 Use of Tools

Three functions of testing tools and three kinds of availability are shown in the table below. Only tools that are generally accepted as practical are shown.

Availability / *Function*	Tool exists commercially	Tool can be developed by routine work	Tool requires research effort to develop
Maintaining a test database	editors	librarians	test database
Utility tools that aid programmers	comparison	dump and edit	self-instrumented programs
Tools that measure or certify tests	path exercise	expression exercise	maintenance coverage

The entries in the table often fall between columns, indicating a range of possibilities. For example, although "dump and edit" programs exist commercially, it may be useful to produce some peculiar to the Command, or to particular systems, a routine programming task.

Each function is described in more detail below.

Tools for maintaining a test data base

Regular file editors and librarian systems that are intended for working on program sources can be used with test data. However, it may be worthwhile to adapt these to the slightly different problem of testing. In particular, while the updating mechanisms of most librarians assume that one release replaces a previous one, for test data two versions often have the same status. Most of the reason for mechanizing the process of test maintenance lies in the potential for building a comprehensive system documentation tool using database inquiry techniques. Such a tool would allow quick answers to questions such as "What tests were applied at level II DCT for the SIDPERS system release of June, 1978, and which test was designed to go with which change request?" Although such an inquiry system is only wishful thinking today, there is an immediate gain in putting test data under a mechanical librarian. It would provide a mechanism for the recording of test decisions at all levels, and insure that test data is not discarded prematurely. Although this is not a problem beyond the level III DCT, it does arise earlier in the testing process where tests are less formal, and programmers pass information from hand-to-hand, or do without.

Utility tools for conducting tests

One measure of success in mastering the art of testing is recognition that routine tools are an indispensible aid. It is a false economy to make do without (for example) comparison programs because each use *can* be done by hand. A number of such tools undoubtedly exist now, and some may be tailored to particular systems. These should be collected, and systematically extended to all systems. Where necessary, comparison programs should be supplemented by low-level dump and edit programs, but whenever exception reporting can be used, it should be. For example, a message like: "files compare exactly except for differing blank fields in records 322, 323, 324" (from a clever comparison program) is far superior to a pair of complete dumps, or even dumps of the differing records. The necessary tools are easily recognized by the programmers who work on particular systems, and where tools are lacking, they are easy to develop or extend. (It should be noted that the effort is a continuing one, however. As tools are used they suggest improvements to themselves, and these should not be ignored.)

Self-instrumented programs do not require much automatic programming suppor[illegible]e necessary features are only those of conditional compilation. However, the decisio[illegible] what built-in testing features to add to a program, and how these should be spec[illegible]d, used, and modified, requires more understanding than is available today.

Tools that measure and certify tests

Path exercisers are now a routine part of the software of many machines. The best compromise between simply checking each statement for execution and exhaustive testing, seems to be "branch testing," insuring that each conditional statement's alternatives are tried. Although this technique has many theoretical failings, in practice it seems to produce good results at low cost [Sorkowitz, 1978]. Furthermore, processors are as available for COBOL as for other languages, and COBOL's wordy structure makes the tools work better. The similar tools that exercise expressions are less well developed, more difficult to apply to COBOL, and more expensive to use. However, their promise seems worth the gamble. The "mutation" system now under development by AIRMICS is both a path and an expression exerciser.

In evaluating exercising systems, there are three factors to consider:

1) Test-data generation *vs.* coverage of supplied test data. In general, the algorithms that generate test data automatically are expensive and not applicable to COBOL programs. However, isolated examples of practical generation algorithms exist (notably, that suggested in [Foster, 1978] for testing formulas).

2) Efficiency and ease of use. Most exercising systems are independent of conventional compilers. The drawbacks are that the systems are interpretive and therefore inefficient by an order of magnitude relative to compiled code; and, since the actual object code produced by compilation is not being used, some nasty surprises (for example caused by compiler "optimization") can appear when the actual code is run. On the other hand, stand-alone systems are much easier to produce independent of the commercial vendor.

3) Ability required for use. The ideal testing tool would generate its own data, use it, and report the necessary program changes to correct errors found. In reality, some exercisers are usable by virtually any programmer, while others require more skill and understanding to use. The more difficult an exerciser is to use, the less its use can be specified as an objective criterion for testing certification; but, better programmers may get better results with sharper tools.

The AIRMICS mutation system rates quite well against these criteria. It judges rather than generates data (but it could use the [Foster, 1978] technique to generate formula data). It is interpretive, but to build it into a commercial compiler would be very difficult. In its research form it is perhaps too difficult to use, and not adapted to audit certification; however, it is not difficult to imagine production versions that do only (say) branch testing and constant substitutions, that would be much better.

The eventual promise of measurement and exercising tools lies in the maintenance situation, where effort can be concentrated on a small fraction of a large program. There are two difficulties, both of which should be the subject of research. First, COBOL is not a language that lends itself to encapsulation of program parts into autonomous modules. The data structure is global, and paragraphs may be PERFORMed, but without parameters. The CALL verb is a candidate for resolving the difficulty, but its use is nonstandard. (It may be that use of CALL should be enforced as part of a stepwise-refinement programming practice within the Command. The topic is outside the scope of this report.) Research is needed on some means to break COBOL programs up after-the-fact, to decompose an existing program. The second difficulty is that exercising techniques do not compose well. That is, when some form of exercising has been applied to parts of a program separately, those parts in combination may not pass a similar test. For example, a subroutine may fail an expression exercising test when it is imbedded in a calling program, because the caller can never pass it data to cover its expressions. While these fundamental difficulties are being investigated, program source comparisons can be used to isolate the changes in program releases, and experiments can determine the effect of exercising applied to the changes alone.

In summary, the Computer Systems Command should be actively encouraging the maximum use of proven, effective testing tools, and at the same time, supporting the conversion of research tools to practical status. A survey should be made of tools available in the Command, and of resources available to extend their use and develop new routine tools. Commercial products should be purchased where available, preferably adapted to special systems. Research and evaluation of new and existing tools should be an ongoing process.

3.6 Practices and Training

In the *development* of large-scale software systems, the role of testing is controversial. The reason lies in functional requirements for new development. State-of-the-art testing tools use program structure to define the coverage of tests, and these coverage measures, while not infallible theoretically, do work well in practice. But they seem to work because they call programmer attention to defects (usually of commission instead of omission) in code. Testing of functional requirements has little connection with code structure, and errors of omission are common. In addition, the best testing tools rely on human beings to supply input data and judge the correctness of output. In the case of new development, outside "users" are ill-equipped to handle the necessary detail, while the inside developer may not fully understand the requirements. Thus although there is agreement that newly developed systems should be tested, there is little agreement on how this should be done, and almost no technical way to evaluate the validity of testing schemes short of massive trial development projects. The alternatives in new development include a wealth of "programming methodologies" that seek to make the software better to begin with, rather than to discover errors after it is ready to test. There is agreement that "random testing," the process of simply trying realistic data and keeping good records about what does and does not work, it not a reasonable way to deal with new program development.

Common-sense testing *is* a reasonable way to handle some maintenance situations, however. Furthermore, the large, relatively stable, long-lived systems in use in the Command provide a unique opportunity to perfect the process of program changes and their testing. Testing is an art, and the general case is very difficult to describe and use. But specific cases can be handled without such a general understanding, and the Command is in the unique position of having a few specific systems, each with a long history and prospective life, and each in extensive use at many installations. If simply keeping good records and paying attention to detail is going to work anywhere, it will work in this situation.

There is a considerable body of knowledge within the developmental agencies about how to test particular software systems. At the final level of the Developmental Center Test (DCT III) much of this knowledge is codified by the required test plans. The developmental agency keeps relatively good test records because it is aware that the situation will arise again and again with the same system. (Each agency is free to develop its own organization to control test data, and the solutions can be expected to differ.) At earlier levels of the DCT, the situation is much more haphazard. People who work on a particular system may see that it is an advantage to control and retain the data and procedures for testing. But without formal guidelines, and with normal personnel movements, it can happen that a programmer required to conduct a level-I test may have to rely on word-of-mouth to get the data and results analysis needed to do the job. Where the necessary expertise is not easily accessable, the result is an inadequate test, which may be exposed later in the testing process, but at great cost. In the worst case, the programmer charged with making an early test may not understand even the most rudimentary rules of program testing, and may think that an inadequate test is perfectly satisfactory, not even seeking out available help.

Codification of existing knowledge and training in its use would benefit all of the DCT levels. Where things are being done well, the high standards could be made universal; where things are haphazard they could be improved. The problems of specifying testing procedures, and of training in those procedures, are closely tied. Training must be widespread to be effective, yet only widely applicable, proven techniques should be taught. The process of developing practices, training in their use, and evaluating their effectiveness is an iterative one that never terminates. It is easy to begin, however, since the Command is large, and much that is done now is effective. A first attempt at a training course can be based on the most conservative methods now in use, and those trained at this level can then be encouraged to suggest and evaluate additional practices. A long process can be shortened by collecting information about current practices, and critically evaluating them for gaps. The initial training could then approximate a final, polished version, and most corrections can be deletions rather than additions. Because knowledge of the existing organization is needed, the data collection effort should be conducted from within the Command (perhaps by Quality Assurance); the critique and initial training program should come from outside.

Specifically,

> (1) The Command should collect the best testing practices now used by developmental agencies, paying particular attention to the recording and documentation of tests (and their links to changes and requirements), and to common-sense practices used at the three DCT levels.
>
> (2) Existing practices should be subjected to critical review to determine where they are deficient. Where deficiencies require new tools or techniques, the cost of purchasing or developing these aids should be noted.
>
> (3) Using the information in (1) and (2), a draft of "testing guidelines" for DCT I-III should be produced. The guidelines should have no force of regulations at the outset.
>
> (4) The guidelines of (3) should be evaluated in trial projects. This will require training of personnel; initially, the guidelines define the training needed. By starting with a limited group of people, the practices, their dissemination, and their effectiveness can be quickly evaluated. Valuable ideas should be more widely propagated through training, more specific guidelines, and eventually regulations and sign-off procedures instigated by Quality Assurance.

The reason a trial-and-error approach to maintenance testing can be expected to work is that the people who conduct tests will be readily able to judge the effectiveness of techniques, if it is in their interest to select and use what works. Unfortunately, the process is neither self-starting nor self-modifying. Any one group, in the process of a particular system change, can easily believe that it cannot afford to use some particular procedure, much less take_time away from the programming itself to develop one. Thus the impetus to keep track of good testing, and to cause its spread, must come from outside of the development agencies. But given that effort must be devoted to tests and their codification, the programmers should be quick to recommend what works and what does not.

3.7 Quality Assurance

Although there is no question but that software can be improved with the use of modern testing methods and tools, at the same time there is no liklihood that errors can be eliminated, or even reduced to an annoyance level. So long as software is written by people using a general-purpose programming language, there will be significant errors that escape notice until too late. To expect anything else is simply unrealistic. Indeed, one reason that a statistical software reliability theory is so controversial is that it predicts that error reduction to any predetermined level is a routine matter, counter to the intuitive feeling of most programmers and programming managers. Given that errors will always be present and always a serious problem, there must be institutional mechanisms to handle them. The people responsible for handling errors and their consequences can and should continually look for ways to put themselves out of business, but should not expect success. They can hope that yesterday's catastrophes become today's minor problems, but not that new catastrophes will not appear.

The Quality Assurance group within the Computer Systems Command is the one charged with handling error problems. Naturally, its official role is the *prevention* of errors, but in practice, program testing is the only device available, and good tests *find* errors, not demonstrate their lack. (The hope that finding a lot of errors means that only a few are left is probably wrong. It can just as well mean that there were an excessive number to begin with.) The prevention of errors is a subject for program design, not program testing. Although an earlier, more specific role in the development process for Quality Assurance may reduce errors at the source, that is not the subject of this report. (Nor is the technology available to suggest that a parallel study be undertaken. There is nothing comparable in development methodology to tools like exercisers. The development methods are subjective, and their validation may never be accomplished. That does not mean that modern programming practices should not be tried; but, the payoff is not so clear as for testing.)

This section is devoted to suggestions that might be called "organizational." However, the constraints outlined in Section 1 apply: the motivation should be technical and the implementation straightforward within the existing framework.

The primary fact of life for any independent monitoring activity is that it will always be understaffed. No enterprise can afford to use more people to check quality than to create the objects to be checked. The solution is to make Quality Assurance an oversight activity. The responsibility for testing does not shift, but the actual people doing the work does.

There are obvious advantages in having software developers conduct tests. First, the developer knows the software (as well as anyone can, even if that is poorly). Since testing is an error-finding process, the developer knows where to look. (The corollary is that it is wrong to reward only *successful* tests—the tester has to be kept honest.) To create an adversary role of developer and independent test organization is to encourage the former to attempt to deceive the latter. In such a contest, the independent, outside group will certainly be the loser. Malicious programming to hide errors is not today a common problem, but procedures should not be developed that might make it so. The second advantage in making the developer perform tests is that it substantially shortens the correction cycle time. Since the developer must make necessary repairs, it is obviously advantageous for him to be there when the problem occurs, thus having the best available information, and cutting short any possible communication and scheduling problems. A third advantage is that responsibility for testing should remain along with responsibility for future maintenance activities. If the developer knows that he must test changes, he is likely to retain the means to do so, along with the system itself. He can evidently do a better job at this than can any outsider, again because of his expertise and the absence of communication difficulties.

The mechanisms that should be institutionalized for testing software depend strongly on its characteristics, and those of the Computer Systems Command are unique. First, the number of field installations is large, but relatively static. Second, those installations have diverse needs but must all use the same software. (The problem of differing hardware resources is now relatively minor, but soon to worsen with two vendors involved.) Third, the software systems are relatively straightforward, but unusual in the vast amounts of data they handle, and in the range of data volumes that occur at different installations. And finally, the systems are long-lived: they tend to be altered to meet new demands rather than replaced. These characteristics indicate that the following situation is dominant in the Command:

> A large, long-standing software system undergoes relatively minor changes, for additional capabilities and error correction at the same time. Despite extensive testing, when released to the field two kinds of errors show up: (1) some of the old functions of the system no longer work properly, and this is observed at almost every installation (but not right away); (2) at a few installations, the new system works very badly (or not at all) because the changes have upset something peculiar to the local operation or hardware.

It should be noted that this situation is by no means the worst that might occur: the software might fail badly at *most* installations rather than a few, and the *new* functions might fail to work properly as well as the old. That this worst case is infrequent shows that present practices have merit, which should not be lost in trying to plug some gaps.

The tools and practices suggested in Section 3.5 and 3.6 are intended to deal with this maintenance situation. The existence of a controlled and well-maintained test data base contributes to discovery of "side effect" errors that the programmer did not imagine, and automatic testing tools show how well the test base is "covering" the new and old code. (It is a demonstration of the effectiveness of those tools if they show that previous tests' coverage has decreased after maintenance—this is an explanation of why changes seem to cause new "random" errors.) Codification of obvious good practices like appropriate file comparisons should insure that there is no slippage in the present standard; in fact, that the best of past practice now becomes the norm. However, the existence of tools and practices is not enough to guarantee that they will be used effectively. The Quality Assurance group must enforce their use. Also, errors will slip through the best testing screen, the more so in the Command's unique situation of serving so many diverse installations, and these blunders must be quickly fixed. The remainder of this section makes specific suggestions to deal with these problems.

Role of Quality Assurance

The Quality Assurance group should become the advocate of testing within the Command. Thus it should be continuously evaluating and developing testing practices, and institutionalizing those that work in the Command's environment. The existing regulations permit this role, although it is less clear that the personnel resources exist to implement it. Quality Assurance could carry out the initial study recommended in Section 3.6.

The process of evaluating and changing testing practices should be a continuing one, and Quality Assurance should attempt to establish mechanisms within the Command to keep it going. For example, there should be an easy way for a programmer who discovers a useful testing method to communicate it to Quality Assurance, and see it adopted if it proves out. Similarly, there should be an open channel to research groups like AIRMICS so that promising new tools can be quickly tried.

At a minimum, the testing practices developed should include test documentation and maintenance tools and methods within the developmental agencies, automatic aids at DCT levels I-III, and objective measures such as specified coverage levels that can be certified formally at the end of development. The training should eventually be a routine part of joining the programming staff of the Command, and all existing programmers should be cycled through it.

Software Maintenance Procedures

When a testing operation is working well, it must still be acknowledged that it will occasionally fail, and corrective action will be required. The procedures that cover release and maintenance of software should seek to minimize the communication and time gaps that exist between official release and correction of an unexpected error. It is easiest to say what *not* to do: the release testing process should not be lengthened in an attempt to duplicate field conditions and thereby release only perfect software. The returns diminish rapidly, because only the field itself can make the ultimate test. The following are recommended:

(1) That level I and II DCT remain the responsibility of the proponent and development agencies, without a formal Quality Assurance role. However, Quality Assurance can serve as technical advisor to the agencies in getting the best results possible. Recommended test procedures should include use of the data recording, comparison, and exercising tools described in Section 3.5. The agencies should be encouraged to trace problems encountered in the more formal DCT III to deficiencies in earlier tests, and to take appropriate corrective measures.

(2) That the existing Environmental Test (ENT) be combined with the existing DCT level III, to be conducted as the latter now is. That is, the developmental agency conducts the test according to a formal plan, and Quality Assurance oversees the activity and certifies its success. Adding the ENT to DCT III means only that some subset of the test is finally performed on a duplicate of the field

hardware, using the software exactly as it is to be released. Such a combination of the two tests should not increase the workload (indeed, because less extensive final tests may be specified, the work may decrease). The benefits will be to retain the developer's expertise (both for the code, and for satisfying function requirements) in the ENT, to cut an entire independent scheduling operation out of the release cycle (even more advantageous if the test is failed and the developer must do corrective work), and to eliminate duplication of DCT III in ENT.

(3) That a new quick-response mechanism be put into effect following the release of software from the combined test of (2). This mechanism would remain in effect from the beginning of the Field Validation Test through a short time period following final release. During this time, problems will be expected, and they are to be handled without recourse to the normal mechanisms. (Notably the full mechanism of the System Change Request and the instigation of another release cycle should be avoided whenever possible.) The ideal is to handle such problems as if they were failures in the level-III DCT, but using the reporting field installation as if it were the test site. This means bypassing the safeguards of developmental and environmental tests, so such changes should be as conservative as possible. However, it is advantageous if the mechanism for making them exists alongside the normal emergency procedures. The quick-response change period would remain in effect until a specified time interval had passed without any emergencies reported. At that time the software release would be considered complete, and a correction would be issued if substantial changes occurred in the quick-response period. (The testing of such a re-release is a problem—the point of diminishing returns occurs early—but it should at least get the minimal environmental test, and there should be at least one site that is known to use it in place of the patched original release.) The quick-response period might best be formalized as part of the Field Validation Test, but since all installation sites would be involved, the authority for ending the period successfully must lie with Quality Assurance and the development agency.

The combined effect of these procedures should be to shorten and restrict the role of pre-release testing to what it can accomplish with efficiency and surety. Then an emergency mechanism would be in place to deal with the errors missed that are bound to occur in the field. In effect, the field is thus acknowledged as the only real test of software, a test that cannot be duplicated beforehand. (It should be noted, however, that any use of the emergency procedures should be carefully monitored to see if the problem could have been caught before release. In the analysis of pre-release test failures, Quality Assurance must play the leading role, to make it in the developer's interest to find out what went wrong.)

Annotated Bibliography

Three collections are often cited:

[ICRS, 1975]
Proceedings of the Second International Conference on Reliable Software (available as the July, 1976 issue of SIGPLAN *Notices*, from the Association for Computer Machinery).

This conference was apparently held at just the right time, for it contains an outpouring of good ideas that is probably unmatched in any other. Not all of them are practical, but almost all of them are well thought out and interesting. Many of the papers have since been reprinted (notably in [Miller and Howden, 1978]).

[Miller and Howden, 1978]
E. F. Miller and W. E. Howden, eds., *Tutorial: Software Testing and Validation Techniques*. IEEE Catalog No. EHO 138–8, 1978.

The title as a Tutorial is a bit misleading, since this is primarily a reprint collection. The editors (who are probably the stars of the testing field) have written some bridging commentary, but it isn't very extensive. The theoretical sections are excellent, and the background presented in the practical sections is also excellent, but some of the papers are showing their age (typically, 1975). Although the weakest parts are those on management and planning, almost nothing else exists in this area.

[IEEE–NBS, 1978]
Digest for the Workshop on Software Testing and Test Documentation, Ft. Lauderdale, Florida, 1978.

As the working document for a workshop organized by E. F. Miller, this proceedings may not be easily available, but IEEE probably has it in the Repository system and so would provide a copy. Some of the papers are scheduled to appear in the *IEEE Transactions on Software Engineering*, but because that journal has a somewhat formal bent, it may fail to include the most practical ones. Although far narrower than [ICRS, 1975], this volume is also filled with interesting ideas, and even a few reports of experience with actual testing tools. Although the title includes test documentation, these papers were the weakest.

The April, 1978 issue of *Computer* is devoted to testing, and contains two survey articles (Huang, Fairley).

The references below are not intended to be comprehensive. (The final section of [Miller and Howden, 1978] is so intended; Miller's company has at times offered to provide the current version of this bibliography--Software Research Associates, San Francisco.)

[Acree et al., 1979]
A. T. Acree, T. J. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward, *Mutation Analysis*, GIT–ICS–79/08, Georgia Institute of Technology, September, 1979.

The most recent presentation of the "mutation" testing idea, including some information about the pilot COBOL system. This report contains many insights into the testing process in the "mutation" framework, sometimes more "for drill" than because that framework is helpful. It is weakest on details of how the systems actually work.

[Darringer and King, 1978]
J. A. Darringer and J. C. King, Applications of symbolic execution to program testing, *Computer* 11 (April, 1978), 51–60.

A tutorial presentation of symbolic execution as a practical testing technique. Its strengths are that whole classes of inputs can be handled at once; its weakness is that the usage is hardly routine, and requires both human skill and a sophisticated computer aid.

[Foster, 1978]
K. A Foster, Error sensitive test cases (ESTCA), in [IEEE–NBS, 1978], 206–225.

A successful application of hardware "stuck at 1/0" fault analysis to generating test data to catch single errors in arithmetic expressions. Some of the examples are in COBOL, and the technique is unpretentious and practical.

[Hamlet, 1977]
R. G. Hamlet, Testing programs with the aid of a compiler, *IEEE Transactions in Software Engineering* SE–4 (July, 1977), 279–289.

The implementation of an expression and path exerciser in a production compiler.

[Heckel, 1978]
P. Heckel, A technique for isolating differences between files, *CACM* 21 (April, 1978), 264–268.

The algorithm presented is not only fast, but is capable of displaying differences in a "natural" way.

[Howden, 1976]
W. E. Howden, Reliability of the path analysis testing strategy, in [Miller and Howden, 1978], 38–45.

The classic analysis of a structural testing scheme, showing why it is untrustworthy. Similar results apply to any exercising scheme.

[Sorkowitz, 1978]
A. R. Sorkowitz, A procedure to improve the quality of program testing, in [IEEE–NBS, 1978], 97–111.

A practical study in which a commercial branch–testing system was used to measure software, applied by an independent certification group at HUD. Its only blind spot is that the testing group did not see the merit in providing the testing tool to the programmers.

Previous studies and manuals:

[SAI, 1977]
Enhancement of Test Procedures, final report for USACSC Fort Belvoir, January 14, 1977. Contract DAHC26–76–D–1004, Delivery Order 0004.

[Daniel and Mitchell, 1976]
R. D. Daniel and J. R. Mitchell, *Present Software Testing Practices* (draft), 1976.

[Fox and White, 1979]
Report of QAD Special Study on Testing, Memorandum ACSC–QA, September, 1979.

[ADR, 1974]
Applied Data Research, Inc., *MetaCOBOL User Guide* Version 7, July, 1974.

MetaCOBOL is quite a good source–to–source processing system that deserves to be used more than it is. The processor is complicated enough that formal training in its use is required, and this does not seem to be available in the Command at present.

[GRC, 1979]
R. Thibodeau, *The State–of–the–Art in Software Error Data Collection and Analysis*, Contract DAAG29–76–C–0100/0598, AIRMICS, January, 1979.

There should be some connection between error taxonomy and program testing, but no one is sure what it is. One practical use of error analysis is to devise "mutant operators" that correspond to errors programmers are likely to make. The difficulty with most data–collection studies is that their procedures are sloppy, and results therefore questionable.